



Article

Multi-Project Multi-Environment Approach—An Enhancement to Existing DevOps and Continuous Integration and Continuous Deployment Tools

Baasanjargal Erdenebat ^{1,*}, Bayarjargal Bud ², Temuulen Batsuren ³ and Tamás Kozsik ¹

¹ Department of Programming Languages and Compilers, ELTE Eötvös Loránd University, Pázmány Péter stny. 1/C, 1117 Budapest, Hungary; tamas.kozsik@elte.hu

² Department of Information and Computer Sciences, National University of Mongolia, Ulaanbaatar 14200, Mongolia; bud.bayarjargal@gmail.com

³ Khan Bank, Ulaanbaatar 17010, Mongolia

* Correspondence: baasanjargal@inf.elte.hu

Abstract: DevOps methodology and tools, which provide standardized ways for continuous integration (CI) and continuous deployment (CD), are invaluable for efficient software development. Current DevOps solutions, however, lack a useful functionality: they do not support simultaneous project developments and deployment on the same operating infrastructure (e.g., a cluster of Docker containers). In this paper, we propose a novel approach to address this shortcoming by defining a multi-project, multi-environment (MPME) approach. With this approach, a large company can organize many microservice-based projects operating simultaneously on a common code base, using self-hosted Kubernetes clusters, which helps developers and businesses to better focus on the product they are developing, and to reduce efforts on the management of their DevOps infrastructure.

Keywords: DevOps; CI/CD; pipeline; Docker; Kubernetes; Helm; dynamic environment; microservices



Citation: Erdenebat, B.; Bud, B.; Batsuren, T.; Kozsik, T. Multi-Project Multi-Environment Approach—An Enhancement to Existing DevOps and Continuous Integration and Continuous Deployment Tools. *Computers* **2023**, *12*, 254. <https://doi.org/10.3390/computers12120254>

Academic Editor: Paolo Bellavista

Received: 23 June 2023

Revised: 15 November 2023

Accepted: 28 November 2023

Published: 5 December 2023



Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

In the modern era, enterprises and organizations place significant importance on DevOps [1,2] and continuous integration/continuous delivery [3] (CI/CD) approaches. These methodologies aim to enhance the agility, productivity, and operational efficiency of their businesses. Moreover, they seek to establish a reliable software development ecosystem characterized by swift turnarounds. DevOps facilitates the convergence of previously isolated roles, including development, IT operations, quality engineering, and security, fostering collaboration to create products of higher quality and reliability [4]. By embracing a DevOps mindset with its associated practices and tools, teams will be able to address customer requirements more effectively, feel confident in the applications they develop, and speed up the realization of business objectives [5]. The implementation of CI/CD practices empowers development teams to deploy code changes promptly and consistently since CI/CD and DevOps share a symbiotic relationship, mutually striving to facilitate the common goal of expediting and streamlining software development [3,5].

By enhancing agility and flexibility, microservices and containers can be an ideal complement to the DevOps ecosystem of an enterprise. With DevOps and microservices, developer teams can create loosely coupled, lightweight, individually deployable services which are easy to maintain, compile and deploy [6–8]. Containers, which are often used in software with a microservice-based architecture, also offer platform independence and interoperability, which align perfectly with the goals of microservice architecture [9]. Each containerized microservice can be deployed and managed independently without being affected by services hosted in other containers [10]. Organizations mostly use Docker containers to package and deploy microservice-based applications because they offer a

simple way to package the code and push it to the release pipeline, which streamlines the DevOps workflow.

A key aspect of DevOps is the way it improves the efficiency of development by enabling automation. DevOps automation requires the use of a technology that can perform tasks with reduced human assistance to processes and that facilitates feedback between the operations team and the development team so that iterative updates can be deployed faster to the applications in production [11]. Tools like Jenkins [12] can be used to automate DevOps workflows by enabling continuous integration and continuous delivery (CI/CD), leading to improved collaboration, minimized manual intervention, and accelerated software delivery cycles.

The research presented in this paper was triggered by a technology change experiment carried out at a large software company that adopted DevOps tools and an agile mindset at the beginning of 2020 and was gradually heading towards a more advanced level of DevOps processes. This paper discusses the results of the experiment and the valuable insights gleaned. In the course of the experiment, most front-end services of the company were migrated into containerized microservices within Docker-based Kubernetes clusters, while the company's core and legal systems were still operating in a traditional way. The fundamental concepts and basic configurations had already been introduced during the initial adoption of DevOps practices and containerization technologies. Over time, as DevOps became more widely used across the company, the number of containerized microservices increased, and new projects are now regularly initiated with DevOps architecture. However, traditional DevOps solutions encounter limitations when it comes to facilitating simultaneous development and deployment of microservices within a shared cluster. This limitation causes delays in product releases, a backlog of projects, and extended delivery times. It also results in challenges such as identifying the causes of failures, increased number of errors, decreased efficiency, and hindered collaboration between teams due to interdependencies, etc. Typically, traditional DevOps workflows involve the establishment of several predefined environments for the deployment pipeline, including stages like DEV, SIT, UAT, and production, and developers deploy their changes into designated environments based on branching. However, this conventional approach imposes restrictions on the concurrent development and deployment of microservices across various projects and teams. Therefore, the original DevOps infrastructure encountered the following range of challenges:

- While it is technically feasible to manually insert steps into the pipeline for each desired environment using existing DevOps tools, this approach is suboptimal and inefficient. The reason is that manually expanding the pipeline in this manner results in an oversized, poorly structured, unorganized, and complex pipeline. Furthermore, it has adverse effects on the functionality and performance of the pipeline, as well as on its potential for future optimization.
- Within the context of continuous integration and continuous delivery (CI/CD), automation plays a pivotal role. Rather than manually adding specific environments to the pipeline, which creates obstacles that impede the smooth flow of the CI/CD pipeline, it becomes crucial to automate the environment provisioning process in a more efficient and advanced manner. Therefore, there is an urge to enhance existing DevOps and CI/CD automation processes to effectively address complex scenarios.
- To facilitate concurrent development across multiple projects, it was essential to find a robust solution. It was imperative to establish separate and isolated environments for each project within a shared Kubernetes cluster to mitigate the risk of conflicts. Therefore, the proposed solution needed to be dynamic and seamlessly manageable within the existing DevOps infrastructure.
- To effectively implement the creation process of multiple environments within the existing DevOps infrastructure, it is crucial to methodically tackle the challenges related to the integration of various DevOps toolchains. This requires a precise

definition of requirements and a careful selection of compatible tools to streamline and optimize the integration process.

- In addition to implementing the MPME approach for simultaneous development, it is essential to adhere to best practices and stay current with advanced technologies. In light of this, the company's existing DevOps implementation required more sophisticated deployment solutions, such as Helm deployment. The complex structures necessitated the incorporation of advanced deployment options.

The main contribution of this paper is the proposal of a new methodology to organize project artifacts to support the multi-project multi-environment (MPME) approach while effectively addressing the aforementioned challenges. With this approach, a large company can organize many microservice-based projects operating simultaneously on a common code base, using self-hosted Kubernetes clusters, which help developers and businesses focus on the products they are building instead of managing their DevOps infrastructure. Our proposal has been implemented and validated in a real-life industrial scenario.

The subsequent sections of this paper are organized in the following way. Section 2 is a brief overview of the background knowledge on DevOps and containerization technologies. Section 3 discusses related work. Section 4 outlines our defined problem statement, while Section 5 expounds on the intricacies of our proposed approach. Section 6 provides the detailed explanation of our customized Helm chart, and Section 7 discusses the pipeline optimization for the MPME. The outcomes of our approach are explicated in Section 8, whereas Section 9 contains the comprehensive conclusions based on our experiment.

2. Background

Modern software development methodologies are distinguished by their frequent and rapid release of software changes, which facilitate immediate feedback from end users. A key driver for organizations embracing DevOps and CI/CD methodologies is the demand for swift, consistent, and automated software delivery immediately after changes have been committed to the mainline. In the past few years, a wide range of tools catering for DevOps and CI/CD practices have emerged in the field. The approach expounded in this paper builds on top of these existing DevOps toolchains. This section concisely outlines relevant DevOps principles and highlights the key tools associated with containerization technologies that were used in the experiment showcased by this study.

2.1. Tools for DevOps and CI/CD Practices

A DevOps toolchain identifies the various tools, technologies, and techniques used to support and enable DevOps practices in software development and delivery. It includes many tools that enable and simplify various stages of the software development lifecycle (SDLC), from code development to deployment and operations [13]. Regardless of the specific DevOps toolchain utilized by an organization, achieving a successful DevOps process requires the strategic use of suitable tools to effectively tackle the key phases of the DevOps lifecycle [14]:

- **Plan**—The initial phase of the DevOps lifecycle focuses on gaining a deep understanding of the project's requirements to ensure the development of a high-quality product. It provides essential inputs for the subsequent development and operations phases. Additionally, this phase helps organizations clarify their project development and management processes. There are various tools, such as Google Apps, Jira, Confluence, Slack, etc., available for this purpose; in our case, we used ClickUp for project planning, tracking, and collaboration.
- **Code**—In the development phase of the DevOps lifecycle, the project is constructed by creating the system infrastructure, writing code for features, and establishing test cases and automation procedures. Developers utilize a remote code repository, such as GitLab or Git, to facilitate code storage, collaboration, version control, and management. Development tools like IDEs such as Eclipse and IntelliJ, as well as

project management tools like Jira, are commonly employed. In our case, we used Git as the version control system.

- **Build**—Applications are built by integrating various codes formed in the previous steps. Frequently utilized tools include Maven, Gradle, Sonatype Nexus, and others. In our case, we employed Maven for this task.
- **Test**—This represents a pivotal phase in the application development process. It involves thoroughly testing the application and, if needed, rebuilding it. We employed tools like Junit, Maven, and Sonarqube for testing purposes.
- **Continuous integration**—This phase automates code validation, building, and testing to verify that changes are implemented correctly and to detect errors at an initial stage. Tools like Jenkins, CircleCI, and BitBucket are employed for this purpose; we utilized Jenkins to establish our CI/CD pipeline.
- **Deployment**—DevOps streamlines the automation of deployment through the utilization of tools and scripts, ultimately aiming to automate the process by enabling feature activation. The code is deployed either in a cloud environment or on an on-premises cluster. The tools commonly used include Azure, AWS, Docker, Heroku, etc. For us, we used Docker-based Kubernetes clusters for deployment.
- **Operations**—This phase involves adapting to changing infrastructure dynamics, which offers opportunities to enhance availability, scalability, and product transformation. Some tools commonly used in this phase include Loggly, Chef, Ansible, Kubernetes, and others. Within our DevOps framework, we employed Kubernetes for container management and Ansible for automating infrastructure and application deployment.
- **Monitor**—Monitoring is an ongoing and integral phase within the DevOps methodology. Its purpose is to continuously monitor and analyze information to assess the status of software applications. The most common tools for this phase include Nagios, Splunk, DataDog etc. We used Prometheus and Grafana for this purpose.

The following key concepts of DevOps and CI/CD practices need to be emphasized, as they constitute the critical components of our MPME approach.

2.1.1. Version Control

DevOps repos encompass a collection of repositories designed for version control and comprehensive management of project artifacts [15]. This facilitation plays a pivotal role in coordinating code modifications across the entire team. In the context of the experiment outlined in this study, Git [16] served as the designated version control system. Git enables meticulous oversight of code, solutions, builds, commits, pushes, pull requests, and branching specifics pertaining to projects. The version control system empowers seamless tracking of every alteration made by each developer, ensuring secure integration of their contributions, rigorous testing, and the controlled release of changes into the production environment [17].

2.1.2. Branching Strategy

A properly implemented branching strategy will be the key to creating an efficient DevOps process. DevOps is focused on creating a fast, streamlined, and efficient workflow without compromising the quality of the end product [18]. A branching strategy helps define how the delivery team functions and how each feature, improvement, or bug fix is handled. It also reduces the complexity of the delivery pipeline by allowing developers to focus on developments and deployments only on the relevant branches without affecting the entire product [17]. The selection process for a branching strategy depends entirely on the developers and the project requirements.

2.1.3. CI/CD Pipeline

A continuous integration/continuous deployment (CI/CD) pipeline is used to accelerate the software development process through automation. It allows organizations to build, test and deploy software projects continuously, making it easier for developers to integrate

changes into the project [19]. Jenkins, which was used in this research, is an open-source CI/CD pipeline for many combinations of languages and source code repositories. It integrates development life-cycle processes of various kinds, including *build*, *document*, *test*, *package*, *stage*, *deploy*, and *static analyses* [3,12].

2.2. Tools for Containerization

2.2.1. Container Runtime—Docker

Docker containers are isolated applications that are broken into smaller lightweight execution environments that contain everything needed to run the code: the runtime, system tools, and system libraries—anything that can be installed on a server [9,20]. It guarantees that the software will always run in the same way, regardless of the environments in which it is developed, or in which it is going to be used in the future [21]. In this study, we used Docker as our container runtime engine.

2.2.2. Container Management System—Kubernetes

Kubernetes is an open-source container orchestration platform that is used for automating deployment and securing and scaling the application while managing and orchestrating containerized workloads and applications [22,23]. Kubernetes supports several container runtimes including Docker. Within a Kubernetes framework, a cluster comprises multiple worker nodes, which are tasked with executing containerized applications, with the primary units of the application workload encapsulated within Pods. These Pods are pivotal components, utilized by Kubernetes to enhance container management. Conceptually, a Pod functions as a logical container, encompassing either a single container or a grouping of closely interrelated containers [24].

2.2.3. Package Manager—Helm

Helm emerges as a highly regarded solution, advocated by industry practitioners, for the effective packaging and deployment of intricate software applications onto Kubernetes-powered clusters and platforms [25]. The functionality of Helm revolves around the deployment of charts where each chart functions as a comprehensive package containing one or more Kubernetes manifests. A chart can incorporate both child and dependent charts, allowing for hierarchical structuring. A chart is a collection of versioned, pre-configured application resources, which should be deployed as a single unit [26].

In practical terms, Helm's charts streamline the process of managing, deploying, and maintaining complex applications within Kubernetes environments, offering a comprehensive solution for managing application configuration, dependencies, and updates. This simplification contributes to enhanced productivity and consistency in deploying software a top of Kubernetes clusters. In this study, Helm plays one of the key roles in implementing the MPME approach.

3. Related Work

With the widespread adoption of DevOps methodology, various practical enhancements and solutions have emerged to address the challenges that arise in various scenarios. Among these challenges, the complexities associated with creating and managing multiple environments have been solved in several ways. In this paper, we will explore known solutions and emphasize their key insights and components for the establishment of dynamic environments within a DevOps infrastructure. Here are the key aspects to consider when implementing multiple environments, as well as potential alternative solutions, including the integration of third-party tools:

3.1. Task Automation

Automation is at the core of DevOps ecosystems as highlighted in several research articles and books, including the DevOps automation book by Duffy [27]. Accordingly, any additional implementations, optimizations, or enhancements within the DevOps workflow

should prioritize automation. We adopted this core principle to provide the level of automation and control that were necessary to manage complex dynamic environments. In this study, we streamlined the automation process and continuous workflow within the CI/CD pipeline, encompassing the tasks of defining, provisioning, and configuring environments. This strategic enhancement serves the dual purpose of reducing manual intervention and minimizing the potential for human errors, thereby ensuring a high level of consistency.

In their books, Been Henry et al. [28] and Brikman Yevgeniy [29] underline the importance of establishing comprehensive parameterized infrastructure-as-code (IaC) templates to allow for easy configuration customization, including instance sizes, network settings, and database configurations. It is also an effective way to manage multiple environments. Moreover, one needs to create automation scripts that leverage these templates and configuration management tools to automate environment creation based on input parameters [30,31]. In our case, we introduced parameterization to Helm templates, enabling the provisioning of multiple environments. Additionally, we developed scripts that were triggered directly from the CI/CD pipeline to facilitate this process.

3.2. Customizing Helm

Helm is mostly used for deploying and packaging microservices and their associated components within a Kubernetes cluster. Its primary advantage lies in simplifying deployment and management tasks, which enhances its adaptability for complex use cases. Many organizations deal with unique applications, thus requiring the creation of custom Helm charts. Helm provides robust support for customization through configuration files [32], and its official documentation [33] offers clear guidance on how to tailor chart templates and values.

The focus of customization frequently revolves around adjusting values, aiming to streamline processes and increase flexibility in intricate scenarios [34]. Throughout our research, Helm also appeared to be a pivotal player in dynamic environment creation [35]. Its customizable hierarchical architecture facilitates the management of multiple environment configurations while allowing for the utilization of shared libraries for them. The manner in which one redefines and customizes a Helm chart is contingent upon their specific objectives, requirements, and the design of their particular implementation.

In our experiment, we incorporated Helm as one of the key components of our MPME approach. We customized our deployment configurations based on the architecture outlined in our target MPME framework, utilizing the flexibility and adaptability that Helm brings to the table.

3.3. Alternative Approach to MPME

One might attempt to address the challenge of simultaneous deployment by employing a temporary and mechanistic approach. This could involve deploying the same code-based applications into a single development environment with distinct prefixes or postfixes (e.g., change1-login-service, change2-login-service, etc.). This concept resembles Kubernetes' default deployment strategy, known as 'Rolling Update' [36], which manages both old and new versions of the application simultaneously until the new version is fully operational, typically by assigning random postfixes [37].

In our experiments, we initially addressed the simultaneous deployment issue using the same approach as described above. This method involved creating separate Jenkins jobs and generating every Kubernetes object with specific prefixes for each microservice. Consequently, the single development environment became cluttered with numerous overlapping, modified microservices that shared the same code base. This situation led to confusion and conflicts, making both the environment and the microservices difficult to oversee and maintain, and created challenges for developers, QA specialists, and even DevOps engineers, who were required to shoulder extra responsibilities in order to manage

the disorganized environment. In our practical use cases, this approach proved to be an unsystematic, complex, and inefficient method.

To overcome these challenges and to establish a more sophisticated, well-designed, user-friendly, and manageable solution, we introduced the multi-project, multi-environment (MPME) approach, which is specifically designed to create a dynamic environment for each active project.

3.4. Integrating Additional Tools

There exist several tools for addressing the strong needs of simultaneous development and deployment in the DevOps landscape. Here, we present some of the tools and provide a short summary of their solutions for creating and maintaining multiple parallel environments.

- A. Qovery [38]—It is an abstraction layer on top of some popular tools (Kubernetes, Terraform) and cloud service providers. The goal of Qovery is to provide a production-ready platform layer which will create preview environments for every pull request. These previews provide early feedback on the application changes before the changes are merged into production. Qovery currently supports multiple cloud service providers including AWS, DigitalOcean, Azure, GCP, and Scaleway. If a company plans to utilize cloud-based infrastructure, Qovery may be a good choice for managing multiple ephemeral environments throughout the software development cycle. However, it is important to note that Qovery is not suitable for private or self-managed DevOps infrastructure setups.
- B. Octopus Deploy [39,40]—It simplifies complex deployments and makes it easy to automate the deployment and operation runbooks from a single place, helping ship code faster, improve reliability, and break down DevOps silos. Even though Octopus Deploy can run on all kinds of infrastructure from on-premises to cloud-native, extra insights and understanding are required for using Octopus Deploy for an existing infrastructure. Some of the additional concepts, such as *target*, *package*, and *packaging standard*, can be a bit complicated if one tries to follow the regular ways. Furthermore, creating additional environments demands meticulous pipeline management, as Octopus Deploy is primarily tailored for predefined environments rather than the dynamic creation and administration of multiple environments.
- C. Humanitec [41]—It enables platform teams to build internal developer platforms (IDPs) on top of their existing stack. The additional configuration, pipeline maintenance, and the core architecture for environment provisioning in Humanitec are different from the ones in our approach. For instance, Humanitec employs two core components, ‘Score’ and the ‘Platform Orchestrator’, which are essential for environment creation. The Platform Orchestrator serves as the central component, generating configurations and deploying applications based on Score workload definitions. Developers use Score files to specify resource requirements in a platform-agnostic manner. The Platform Orchestrator interprets these definitions and provisions the necessary resources. Score itself consists of two components: the Score Specification, which outlines how to run a workload and the Score Implementation CLI, which serves as a conversion tool to generate platform-specific configurations. To integrate Humanitec into the development pipeline, one needs to configure pipelines with variables and secrets, set environment variables for new environment configuration in the Platform Orchestrator, install the score-humanitec binary into their pipeline runner, make a call to the Humanitec Platform Orchestrator API within the pipeline to create a new environment, and finally, configure the pipeline to execute the score-humanitec CLI to initiate the deployment. While this approach introduces additional concepts and configurations, it can be a valuable choice for organizations open to this extension in their DevOps toolchain.
- D. BunnyShell [42]—It supports on-demand or automatic creation of production, development, and staging environments. The process of creating an environment on Bun-

nyshell requires choosing between two methods: a Helm chart or a docker-compose. The Helm chart approach on Bunnyshell is conceptually similar to ours, as they employ a 'bunnyshell.yaml' file for environment definition. In contrast, their approach to docker-compose involves the implementation of environment variables at three levels: Project, Environment, and Component. These environment variables are inherited automatically by all applications, services, and databases within the Environment, which distinguishes it from our approach. BunnyShell offers seamless integration with Kubernetes clusters from major cloud service providers like AWS, Azure, and DigitalOcean, in addition to options for private Kubernetes clusters. They provide a range of solutions for automatic environment creation, including Helm chart, docker-compose, and terraform. Choosing BunnyShell might be advantageous for companies seeking a different approach to ephemeral environment provisioning.

- E. GitLab CI [43]—It is a robust solution for automating software development and deployment tasks, providing features such as build automation, testing, version control, and collaboration. However, it has some limitations when it comes to handling multiple environments, especially dynamic ones. The primary challenge lies in maintaining distinct configurations for each environment, involving the setup of various pipelines, variables, and secrets. While GitLab CI does support dynamic environments through 'review apps', its design primarily focuses on managing individual applications rather than a collective group. To address this limitation, tools like Qovery and Bunnyshell, mentioned above, offer integration options with GitLab to simplify the management of multiple environments. These tools can be particularly valuable if developers find GitLab's dynamic environment management too complex for their specific needs. Ultimately, the decision to use GitLab should be based on the company's preferences and their intended approach to managing their development and deployment workflows.

Developers have the flexibility to select their approach for addressing challenges related to simultaneous deployment. However, there is a crucial factor to consider: the alignment of their choice with their specific requirements. These considerations include various factors such as the use of a cloud service provider, the willingness to incorporate additional tools into the existing DevOps toolchain, associated costs, compatibility with particular frameworks and systems, and the availability of experts for solution design and implementation. Based on these factors, one can decide to either integrate one of the tools mentioned above or develop their own customized solution based on core DevOps principles and best practices.

It is important to recognize that these tools may not be sufficient to address emerging challenges in various scenarios. In our case, the aforementioned tools were still unsatisfactory to fully address the simultaneous deployment challenges due to compatibility concerns and the company's own preferences. Consequently, we designed and implemented our novel MPME approach, capable of addressing these challenges without the need for third-party tool integration, all while relying on industry best practices. Our paper provides a comprehensive explanation of the implementation of the MPME approach, which was developed to overcome the encountered challenges related to the concurrent deployment of multiple projects.

4. Problem Statement

Enterprises and organizations frequently manage multiple concurrent business initiatives. The simultaneous development of a growing number of microservices and projects requires more intricate structures within the DevOps framework. Developers from various teams often collaborate on projects and share workspaces. Therefore, development cycles must maintain a continuous and iterative flow, supported by DevOps and CI/CD practices, while bugs and errors must be addressed swiftly [44,45]. In the traditional DevOps infrastructure, environments are becoming more complex and with greater fault tolerance. All developers and QA specialists use environments; however, the limitations on shared environments result in only one of them being able to deploy to a pre-production environment

at any given time. This situation entails waiting or competing for access to environments when developing and testing new features. Therefore, without proper enhancement or integration of additional tools, the traditional DevOps is inadequate for supporting simultaneous development and deployment of the same code-based application within a shared environment. This limitation not only creates bottlenecks for ongoing releases but also might result in a backlog of projects, thus extending the expected product delivery time. This challenge is further compounded by the difficulty in pinpointing the root causes of failures, heightened proneness to errors, reduced efficiency, and poor collaboration due to interdependencies between teams.

To overcome these challenges and improve the landscape of concurrent development and deployment, several obstacles must be addressed.

- Existing branching strategies and pipeline operations only allow us to deploy an application into a single environment, and limit our opportunities for simultaneous development and deployment.
- Well-defined automation is required for environment management to avoid manual labor for operations such as creating an environment and setting up basic configurations for it.
- The existing CI/CD process relies on static parameters designed for predefined environments, including development (DEV), system integration testing (SIT), and user acceptance testing (UAT). Our aim, however, is to implement a dynamic framework that can automatically provision and configure multiple environments (multiple DEV environments and multiple SIT environments) based on the active projects.
- Complex scenarios, like the implementation of the MPME approach for concurrent development, demand more advanced and meticulously organized deployment solutions. It is crucial to embrace best practices such as Helm for deployment and to stay current with cutting-edge technologies.
- To establish multiple environments efficiently within the current DevOps infrastructure, it is essential to methodically tackle the challenges posed by integrating various DevOps toolchains. This requires a precise definition of requirements and a careful selection of compatible tools to streamline and optimize the integration processes.

To effectively address the aforementioned obstacles, we have introduced the multi-project, multi-environment (MPME) approach. This novel methodology is designed to seamlessly integrate with existing DevOps infrastructures. It supports the automated on-demand creation of environments, and thus it will be easier to streamline the concurrent development and testing across a spectrum of projects. With the MPME approach, all the components that are required to build the dynamic environments come as a service.

5. Multi-Project Multi-Environment Approach

The results presented in this paper were implemented and validated in a real-life industrial scenario. The organization engaged in this project has a workforce of approximately 120 developers. Their operational landscape hosts over a hundred front-end products, all functioning as microservice-based container components integrated into their DevOps infrastructure. This operational environment experiences an average of 240 planned and emergency changes per month. Consequently, the yearly number of changes reaches a significant count of over 2800, primarily centered around service improvements and product developments. Moreover, a noteworthy average of eighty projects is announced annually. All these dynamics lead to the continual involvement of development teams, engineers, and QA specialists in parallel and concurrent tasks. They frequently transition from one project to another due to the sheer volume of project inquiries and ongoing endeavors. Consequently, the importance of promptly and automatically provisioning environments in response to active projects becomes notably significant.

We have integrated the MPME approach into the pre-existing DevOps infrastructure to bolster its capabilities for concurrent development and deployment. Our implementation involved utilizing a non-production Kubernetes cluster, composed of a single master

node and ten worker nodes, which were established as an integral part of the automatic workflow within the DevOps infrastructure. To demonstrate the effectiveness of the MPME approach, we developed Java-based front-end applications and microservices concurrently, catering to diverse project requirements. These applications were executed within multiple environments, which were seamlessly provisioned through the MPME approach on that Kubernetes cluster. The principles and methodologies detailed in this study were executed under the following setup.

- Master node: n1-standard-1 (16 vCPU, 32 GB memory)
- Worker nodes: n1-standard-2 (32 vCPUs, 125 GB memory—for each)
- Networking: calico-3.19.1
- Kubernetes Version: 1.21.3
- Docker Version: 20.10.4
- Helm Version: v3.8.0

Our container-based setups rely on *Docker* and *Kubernetes*. In the realm of the DevOps workflow, we employ *Git* for source code versioning, *Maven* for project building, *JUnit* for testing, *Azure DevOps* for repository management, and *Jenkins* as the CI/CD pipeline.

5.1. Architecture in the MPME Approach

The architecture for the multi-project, multi-environment approach allows multiple development environments and multiple QA environments in a self-hosted Kubernetes cluster. We have used specific project-based name components in environment names to distinguish different uses and project teams. The number of the used project-based name components—and hence the total number of environments—can be set based on the available infrastructure capacity. Since we use Jenkins as a CI/CD pipeline, each environment is created automatically using a Jenkins job. The input parameter for this job is the environment name. This job creates and configures all the resources required for a new environment. Once created, microservices can be directly deployed to a particular environment. Figure 1 illustrates the architecture of the MPME approach.

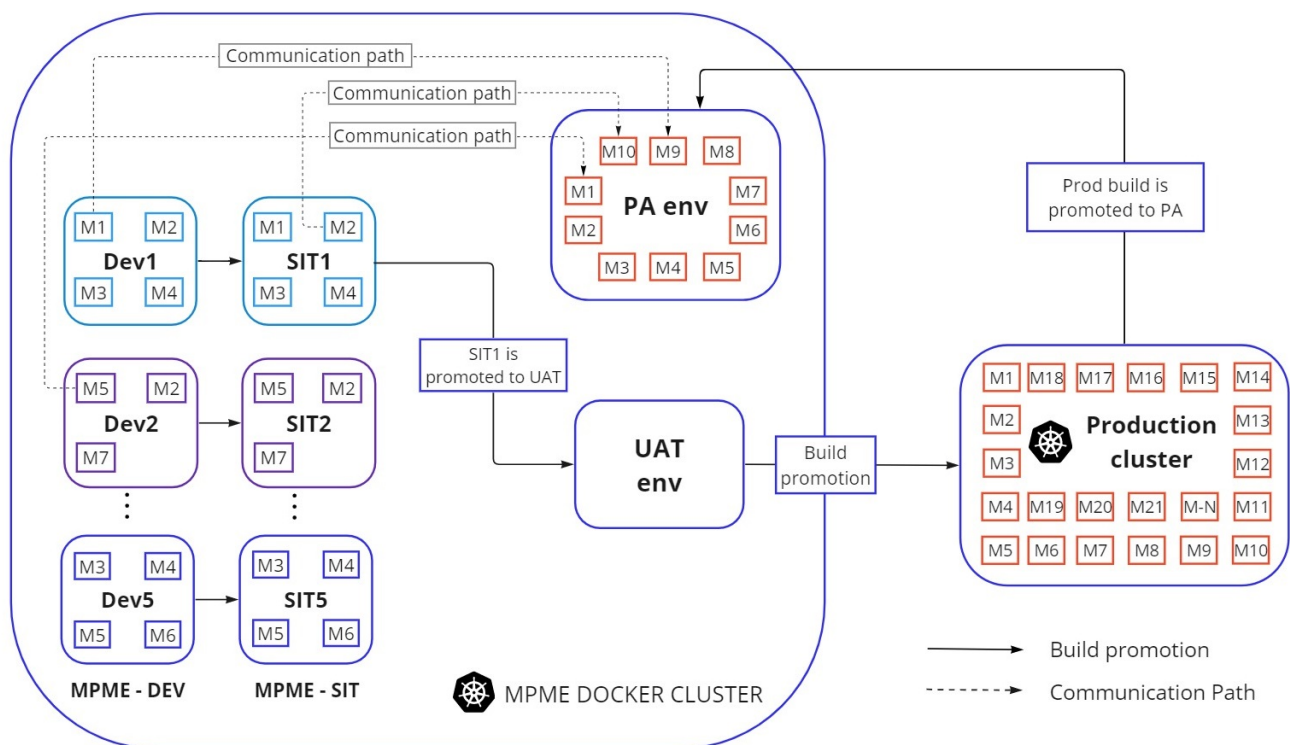


Figure 1. Architecture of the MPME approach.

In this figure, 'M(n)' represents a microservice, 'Dev(n)' is an environment used by a given development team, and 'SIT(n)' is an environment assigned to a quality assurance (QA) team for integration testing. These Dev and SIT environments contain only those microservices which require a code change. Stable versions of all other microservices can be found in a separate environment called 'PA', which stands for 'publicly available'. This stable environment holds the post-production deployments for production-like testing purposes. Additionally, UAT is a staging environment where the functionality of the microservices is checked before production deployment. Each of the above environments can be located in the MPME Docker cluster. In our experiment, we used a separate cluster for production code. The MPME approach defines how environments for development and testing shall be created and deleted. Once the development and testing activities have been completed and the project has been released to production, the environments will be destroyed by a Jenkins job, which deletes all related resources automatically from the cluster.

One of the crucial parts of the MPME implementation is the branching strategy, which is tightly linked to the environment creation process. The *feature*, *develop* and *master* branches had already been set up for the previously used DevOps infrastructure, and now a *release* branch is introduced specifically for the MPME approach, as shown in Figure 2. The branches [46] and their purposes are as follows.

- The *feature* branch is utilized in the development of new features involving specific microservices. Any Jenkins build originating from a *feature* branch should be directed into a DEV environment of the cluster.
- Once all changes are finalized within a *feature* branch, they are merged into the corresponding *develop* branch, thus forming the pre-production code.
- The *release* branch is strategically designated for the upcoming release cycle, exclusively accommodating bug fixes or configuration adjustments. Derived from the *develop* branch, these *release* branches enable the quality assurance (QA) team to rigorously assess the current release. After confirming its readiness, the *release* branch is merged into the *master* branch and distinctly tagged with a version number.
- The *master* branch serves as the main branch intended for all the production code. Once the code in a *release* branch is ready to be released, the changes are merged with the *master* branch and used in the production deployment.

To implement the MPME approach using the Git branching strategy, a name component like alpha, beta, gamma, ..., pi, etc., can be incorporated into the name of a *feature* branch. For instance, 'dev-pi-login-endpoint1' is a *feature* branch name for project team 'pi'. The prefix 'dev' indicates that this *feature* branch must be deployed to one of the development environments. The name component 'pi' denotes that this feature is designated for deployment in the 'dev-pi' environment of the cluster. Staying in line with DevOps best practices to build once and deploy multiple times [47], the build is created from the RELEASE branch and propagated across till production.

Once developers complete the several development steps of the microservices in a single project such as 'login-enhancement', they will raise a pull request (PR) to merge the changes to the *develop* branch. Then, a *release* branch needs to be created from the *develop* branch for the full-cycle test. The *release* branch name should start with the project prefix 'pi', and end with the keyword 'RELEASE', for example: 'pi-login-enhancement-RELEASE'. This *release* branch can deploy the microservices from the SIT environment to the UAT environment after the admin approval. We added an approval step before the deployment of a microservice, so we can prevent incomplete changes from going to the production or development environment. The environment yml files within the Helm chart (more explanation of yml and Helm is provided in later sections) contain specific configurations for the environment, like 'sit-pi.yml' for one of the SIT environments, and 'prod.yml' for the production environment, etc. The same configuration file is used for the creation process of the environment as well. QA specialists will work on testing when the microservices are deployed into a 'SIT' environment. If the changes pass the testing stage, they can be

deployed into the production environment after the current *release* branch is merged into the *master* branch. It is important to merge back into the *develop* branch because critical updates may have been added to the release branch, and they need to be accessible to new features.

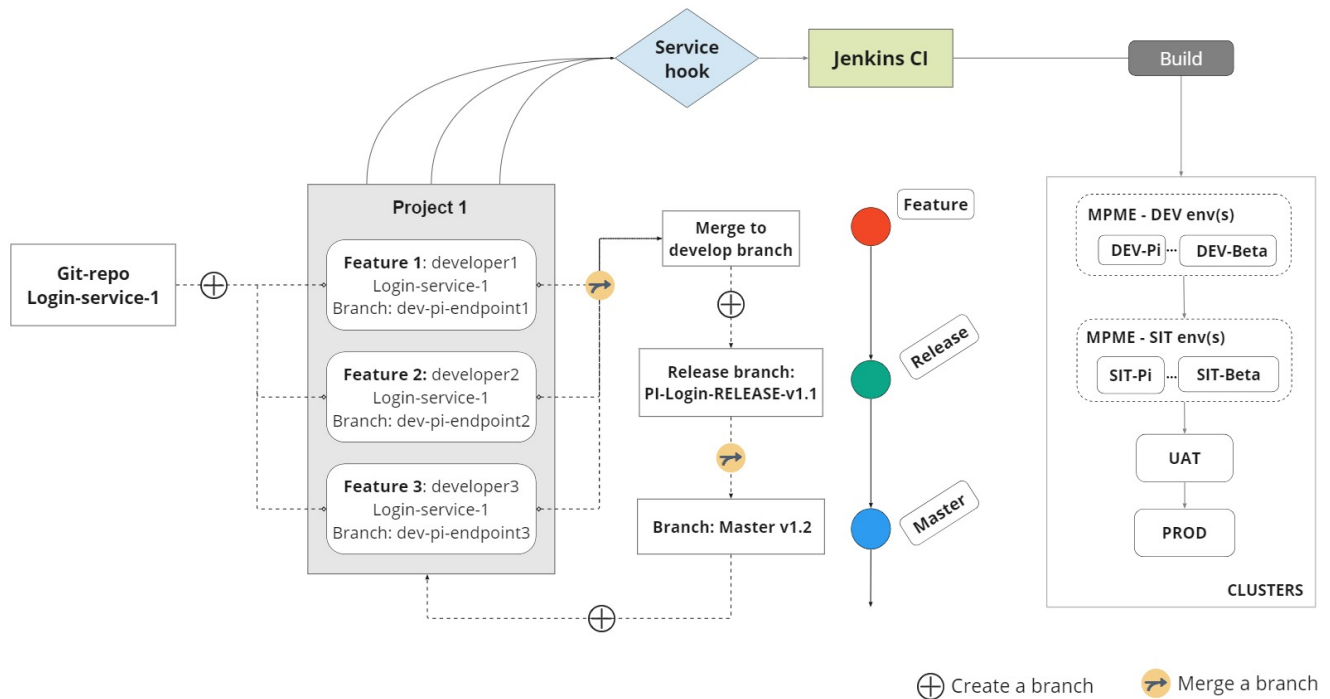


Figure 2. Workflow in the MPME approach.

The MPME approach furnishes a comprehensive solution for managing diverse dynamic environments within the current DevOps ecosystem while fully incorporating the fundamental principles and workflow of the software development lifecycle in DevOps and CI/CD practices. Here is a detailed breakdown of how the MPME approach operates within the context of a DevOps workflow cycle:

1. **Initialize Parameters:** To create environments for active projects, the first step is to define crucial parameters. These parameters include the project name, branch name, environment type (e.g., development, testing, production), and the necessary configuration values. To implement the MPME approach, it is essential to create a feature branch that incorporates the *project name component*, such as 'dev-pi-login-enhancement'.
2. **Development:** The next step focuses on necessary code modifications and development activities for the microservice. Developers must monitor vigilantly the changes in the version control repository, such as Git, where the code is hosted.
3. **Creating environment with MPME:** Developers need to initiate the build command, which in turn triggers a Jenkins job to verify the presence of the environment based on the specified 'project-naming component'. If the target environment is not found within the cluster, this job takes over provisioning the necessary resources. This provisioning encompasses tasks such as configuring networking, security policies, and essential environment-specific settings. Moreover, the job applies the relevant configuration values and secrets to ensure the environment is appropriately configured.
4. **Deploying microservices:** Once the environment has been successfully created, the workflow advances to deploy microservices within that specific environment. The MPME approach utilizes either Helm deployment or the traditional kubectl deployment, depending on the presence of Helm folders. This deployment process ensures

that microservices are effectively deployed into the newly created environment, complete with the required Kubernetes resources and configurations.

5. **Running tests:** Once the microservices are up and running, commence an extensive testing phase to validate the proper functioning of both the environment and microservices. They evaluate functionality and perform testing of code changes with client microservices located in the stable environment of MPME. This testing phase includes a range of test types, such as unit tests, integration tests, and end-to-end tests, as necessary.
6. **Monitoring and Validation:** It is essential to maintain continuous monitoring of the health and performance of microservices within the designated environment. This includes verifying smooth communication among microservices and ensuring the overall stability of the environment through ongoing validation.
7. **Cleaning up the environment with MPME:** Upon successful project completion, or in alignment with project lifecycle or policy guidelines, developers must trigger an automated cleanup process. This process, which includes the systematic removal of the project's host environment, along with all associated Kubernetes resources and configurations, is effectively managed through an automated Jenkins job within the MPME approach.

This concludes our high-level discussion of the main architecture and workflow used in the MPME approach. The following subsections cover the environment creation process, the implementation of communication paths within a specific environment, and isolation of environments.

5.2. Environment Creation Based on Project

In the MPME approach, environment creation is based on the branching model, and uses a project-based naming convention. For instance, if the developer executes the build command from the branch named 'dev-alpha-login-change', the corresponding 'dev-alpha' environment will be generated. Then, the change will be deployed into the newly created 'dev-alpha' environment of the MPME cluster. The branches, environment names, and project-based name components are closely connected in an interdependent way in order to support the creation of multiple dynamic environments for each project.

For creating environments on a Kubernetes cluster, we introduce an appropriate section in the deployment script, as seen in Figure 3. It checks the existence of the target environment in the cluster before any microservice is deployed into it. If the environment exists, the deployment section of the script will be triggered, and the microservice(s) will be directly deployed into the target environment.

```
kubect1 get ns $1 --no-headers -o custom-columns=:metadata.name
result=$(kubect1 get ns $1 --no-headers -o custom-columns=:metadata.name)
if [ "$result" == $1 ];then
echo "Namespace already exists"
else
echo "Namespace not found. Creating now... "
curl -k -X POST
https://user-acc:119e51073xxxxx@jenkins.local:443/job/mpme-
env/buildWithParameters?envName=$1
sleep 90
fi
```

Figure 3. Environment creation section in deployment bash script.

If the environment does not exist, a Jenkins job called 'MPME-env' is used to create it through a *curl* request. While the environment is being created, the following configurations must be set for each environment in order to safely run the privileged microservices.

1. **Secrets**—These contain sensitive information/data that the microservice needs to run, such as passwords, tokens for connecting to other pods, and certificate keys. Using a Secret means that one do not need to include confidential data in the application

code. Because Secrets can be created independently of the Pods that use them, there is less risk of the Secret (and its data) being exposed during the workflow of creating, viewing, and editing Pods [48]. We must provide the following custom secrets for an environment:

- (a) PFX Certificate Files: These files encapsulate certificates, public and private keys tailored for the microservice;
 - (b) Java Key Store (JKS) Files: These files contain a collection of encrypted keys and certificates that ensure the proper operation of the microservice.
2. Configuration service—In the microservice world, managing configurations of each service separately is a tedious and time-consuming task [49]. Within our Kubernetes cluster and microservices ecosystem, we have implemented centralized configuration servers, effectively streamlining the configuration management process. Each microservice is empowered to retrieve their specific configuration from these dedicated servers. Notably, we have established two distinct configuration servers: one exclusively for mobile-oriented microservices, and the other catering to web-based ones. Consequently, the following configuration services are indispensable for each environment:
 - (a) config-service-1: Tailored to accommodate web-based microservices, this service assumes a modified form to align with the MPME approach;
 - (b) config-service-2: Designed to cater to mobile-based microservices;
 - (c) Logging configuration: A configuration that governs the logging of the microservices.
 3. Routing external traffic to environments—Upon establishing a designated environment within the cluster, it becomes imperative to activate and inject Istio [50] into the environment. This step empowers us to precisely direct external traffic to the intended Pod or microservice housed within the target environment.
 4. RBAC (role-based access control) policy implementation—In the context of establishing multiple environments within a single cluster, the security boundary is defined by a namespace. A *service account* [51] has access to only that particular namespace with restricted RBAC [52] policy. The namespace then represents the environment, and the limited scope (access control) of the *service account* prevents modifications of the Kubernetes resources/objects of any other environment while it allows deployments only to that specific environment. The *service account* and RBAC policy will be automatically applied to the target environment. So, the following three Kubernetes resources must be created:
 - (a) A *service account*—This allows us to manage the deployment of microservices and their resources on the cluster through a defined role and role binding;
 - (b) A *role*—This entity defines permissions within a specific environment. While creating a *role*, it is necessary to specify the environment to which it pertains [53]; and
 - (c) A *role binding*—A *role binding* grants the permissions defined in a *role* to a user or a set of users; it holds a list of subjects (users, groups, or service accounts), and a reference to the *role* being granted [53].

When the environment is successfully created through the pipeline, the above ‘must-have’ configurations will be automatically applied on it, and microservices can go up and run without any issues. To reduce human interaction with the configuration-related tasks, a default configuration set is provided, and the pipeline is optimized accordingly.

5.3. Facilitating Communication Paths across the Environments

DEV and SIT environments contain only the microservices that are being developed or modified in a single unit of work (e.g., the introduction of a new feature). Containers of those microservices can communicate with each other using the *localhost* address in the local environment by default. We need to provide, however, a communication path across the environment boundaries as well, specifically to the ‘publicly available’ (PA) environment, which

holds prod-like stable microservices. Therefore, the use of the concept of external domain names (*ExternalName*) is proposed (see Figure 4), which is one of the *Service* types of Kubernetes.

A *Service* is an object (the same way as a Pod or a Secret is an object) of Kubernetes, which is a logical abstraction for a deployed group of Pods in a cluster [54]. With a *Service*, the application/microservice gets a stable IP address that lasts for the lifetime of the *Service* (since otherwise Pods are ephemeral and their IP addresses can change frequently) [55]. Among the five available *Service* types in Kubernetes, two are of particular relevance to our case. The first is the *ClusterIP* (default) type, which enables internal clients to send requests to a persistent internal IP address. The second, the *ExternalName* type, is vital for our purposes. This type allows internal clients to employ the DNS name of a *Service* as an alias for an external DNS name [56]. Specifically, the *ExternalName* service permits the routing of traffic to an external service without necessitating the definition of selectors, specific ports, or endpoints [57].

Upon the environment's creation, an automated Jenkins job integrates "Service" references into the microservices of the 'PA' environment, as illustrated in Figure 4. However, when a particular microservice is deployed into the target environment, it is necessary to remove the corresponding external "Service" reference to prevent potential conflicts between the local microservices and external ones.

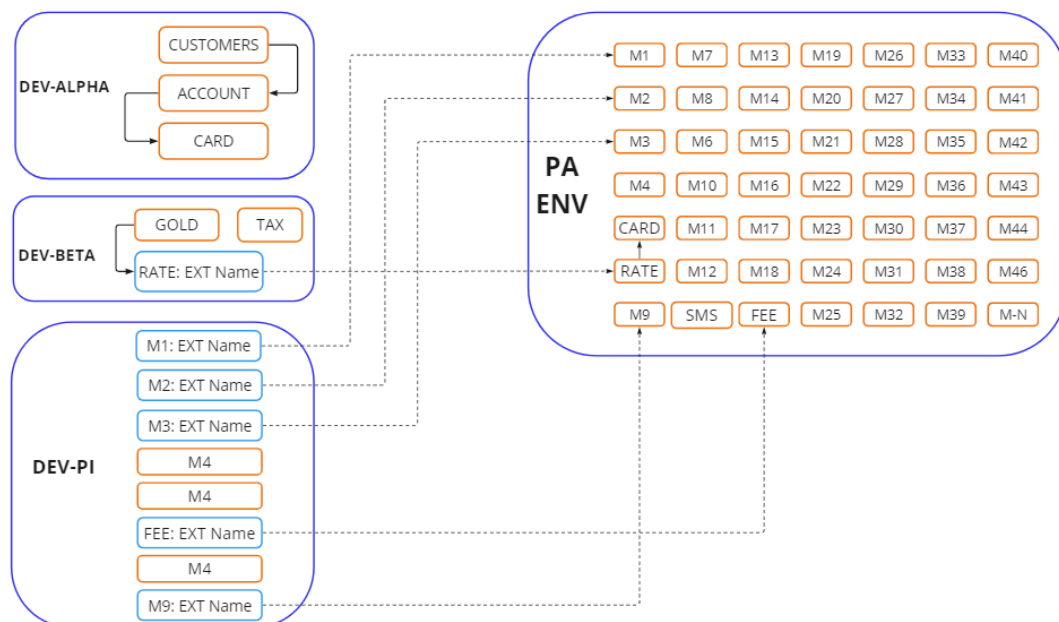


Figure 4. Communication path between MPME and 'PA' environment.

To address this, a script (depicted in Figure 5) can be utilized. This script performs a critical task: it detects the type of the *Service* and removes the *ExternalName* service if present, before the deployment procedure of the microservice starts. This preemptive action prevents any clash between the local microservices and the external services. This action facilitates the creation of a *ClusterIP* service [58] under the name of the deployed microservice, ensuring smooth local communication within the environment.

As mentioned earlier, our proposal introduces a stable environment labeled 'PA' within the MPME cluster. This environment serves as a repository for post-production deployments, housing the latest stable releases of microservices. The 'PA' environment significantly enhances testing capabilities by providing a platform for comprehensive end-to-end tests using production-like microservices. Furthermore, this approach contributes to optimizing resource utilization by eliminating the need to deploy essential stable microservices in every provisioned environment.

```

#!/bin/sh
echo "Checking for External names..."
kubectl get svc $1 --no-headers -o custom-columns=:spec.type -n $2
result=$(kubectl get svc $1 --no-headers -o custom-columns=:spec.type -n $2)
if [ "$result" == "ExternalName" ];then
echo "External Service name exists. Deleting external name..."
kubectl delete svc $1 -n $2
echo "Deleted external name"
else
echo "External Name does not exist. Service can be deployed."
fi

```

Figure 5. Bash script fragment for removing External Name.

Consider a scenario where microservices are distributed across various DEV environments within the MPME cluster, as depicted in Figure 6. Each microservice corresponds to a distinct part of development. For instance, the M1 microservice resides in the “dev-alpha” environment, while the M2 microservice is deployed in the “dev-beta” environment. If M1 in the “dev-alpha” environment needs to communicate with M2, but M2 is not present in the local environment, then M1 must discover the M2 from the stable environment. To prevent locating M2 from improper environments like the “dev-beta” or “dev-gamma”, which hold underdeveloped versions of M2, careful measures should be taken to enable communication only with the stable ‘PA’ environment.

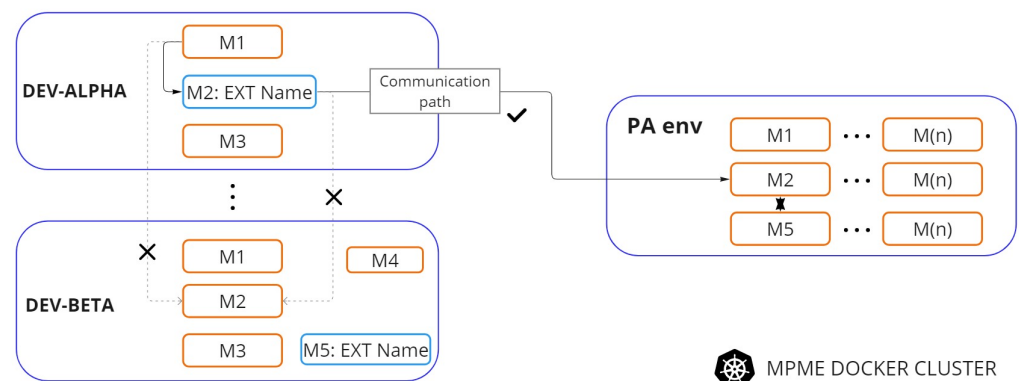


Figure 6. External Name pointing.

To sum up, each environment can house multiple identical microservices originating from distinct teams and projects, each engaged in diverse developmental activities. When QA specialists undertake comprehensive testing for a specific microservice, it is imperative to establish communication with the stable version of the target microservice. This methodology greatly diminishes the risk of production deployment failures, unexpected service interruptions, errors, and ultimately safeguards the quality of the end-user experience.

5.4. Isolating the Environments in a Shared Cluster

The MPME approach ensures isolation and security through containerization, which involves segregating the microservices into distinct containers within their respective project-specific development environments. Furthermore, we implement security measures such as access controls, network segmentation, and encryption in the shared infrastructure when multiple projects are involved. In containerization technology, environment isolation is achieved through cgroups and namespaces [59]. The MPME approach aligns with this same level of environment isolation seen in containerization. Each project operates within its dedicated environment, represented as a namespace within the Kubernetes cluster. This stringent isolation ensures that projects remain autonomous and secure within the shared cluster environment, effectively preventing conflicts and ensuring consistent performance.

6. Customized Helm Deployment Strategy for MPME

Helm, a Kubernetes package manager, streamlines the deployment and administration of Kubernetes manifests and applications by enabling the creation of comprehensive packages that can be seamlessly promoted to the cluster [25]. This tool furnishes a structured approach to define, install, and upgrade applications within a Kubernetes cluster while facilitating the management of versioned templates. The core unit of Helm packaging is called a *chart*. A chart is a collection of files that describe the Kubernetes resources required to run an application. It typically includes a set of Kubernetes YAML [60] files for deployment, services, configuration maps, and other resources [61,62]. Leveraging Go templates, Helm efficiently generates Kubernetes manifests from the chart's templates. Templates allow one to parameterize the deployments and make them configurable and customizable. By using values, one can provide input parameters to the templates, facilitating customization of deployments to meet specific needs [61].

Implementing MPME introduces inherent complexities, particularly in the autonomous management of configurations, secrets, access policies, and Kubernetes manifest files for distinct projects and their respective environments. While Helm can address these challenges, its potential can be fully utilized through customization. Consequently, we have introduced a tailored Helm chart that capable of supporting dynamic environment creation. Each environment can have its own set of configuration values, and the deployment templates will select the appropriate values for a particular environment. This allows us to easily manage multiple environments with different configurations using a single Helm chart. We redesigned the hierarchical structure in which Helm stores the configuration values, with a microservice at the top, and the various environments organized below. This structure is illustrated in Figure 7. The project repository of a microservice may contain a Helm chart folder, and a K8S folder within it; in such a case, the microservice can be deployed into a Kubernetes cluster through Helm deployment.

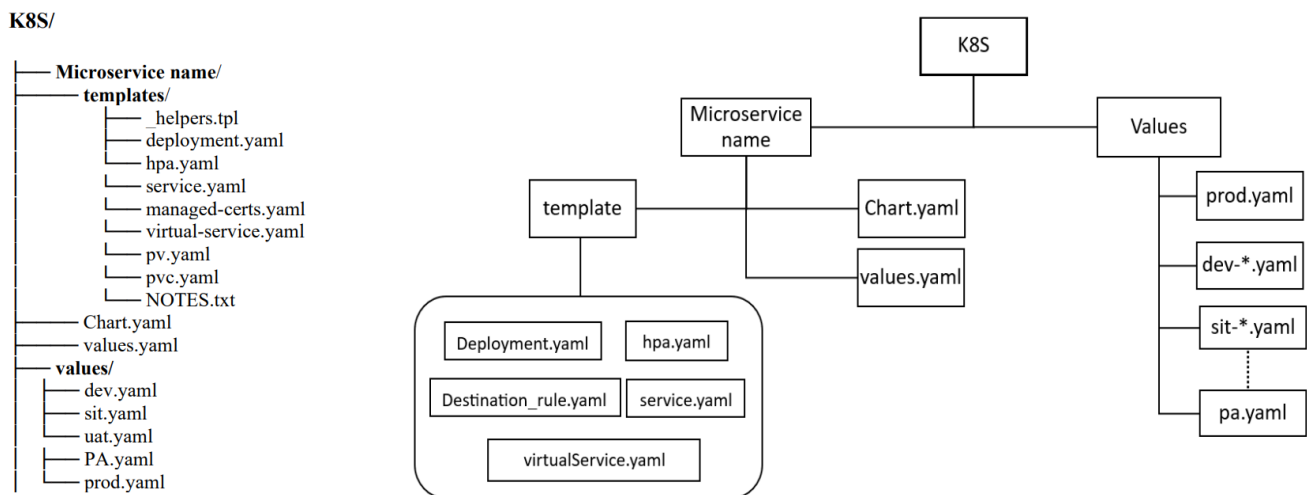


Figure 7. Customized hierarchical structure of Helm for MPME approach.

Values files are crucial for Helm's install/upgrade commands [63]. To minimize redundancy and the need to make changes in the values at multiple places, we split the *values* file into levels: we place a *values* file at the root level, and we provide environment-specific values at the environment level, such as in *values/prod.yaml*. Ports and probes that are common to all environments should be specified at the root level. Entries with default values can also be provided here, and possibly be overridden in the environment-specific *values* files. This hierarchical approach ensures consistency while allowing for environment-specific configurations.

Figure 8 shows the basic commands for Helm deployment. These can be run with a hard-coded path to the environment configuration file.

```
$ helm install --debug --dry-run -f k8s/values/dev.yaml
./k8s/service_name
$ helm upgrade --install ./k8s/service_name -f k8s/values/dev.yaml
```

Figure 8. Helm deployment command of the DEV environment.

To enhance the flexibility and adaptability of the workflow, it is imperative to eliminate any hard-coded paths and fixed parameters. As a solution, we propose a more versatile Helm deployment command, illustrated in Figure 9, that harmonizes with the dynamic nature of the MPME approach.

Our proposed customization to Helm introduces the capability of deploying a microservice across various environments with distinct versions and configurations, all achieved through a single build command within the CI/CD pipeline. This advancement reduces the necessity for manual command execution for each microservice configuration, streamlining the process. Moreover, in cases where a microservice necessitates interaction with persistent storage for data reading or writing, additional enhancement can be effortlessly implemented. This enhancement entails extending the configuration of the Helm chart to encompass the creation of a *persistent volume* (PV), a storage resource within the Kubernetes cluster [64], as well as a *persistent volume claim* (PVC), representing a storage request from a user or service [65,66]. Additionally, the Helm chart can be configured to automatically mount the PV as required.

```
$ helm install --debug --dry-run ./k8s/${env.serviceName} --
set=image.tag=${env.branchName}-${env.version}-${env.BUILD_ID} -n
${stage}
$ helm upgrade --install -f ${files[i]} ${helmValues.nameOverride}
./k8s/${env.serviceName} --set=image.tag=${env.branchName}-
${env.version}-${env.BUILD_ID} -n ${stage}
```

Figure 9. More flexible command for Helm deployment.

To support the MPME approach (including the automatic creation of PV and PVC, and the deployment of microservices with various configurations, etc.), changes have been proposed in Helm charts, and integration within the pipeline has been worked out. Furthermore, the configurability of the pipeline has been considered to allow for the choice between utilizing *Helm install* or *kubectl install*.

7. Pipeline Optimization

DevOps and CI/CD practices allow companies to develop software faster and to make it deployable for production in a short time frame [67]. The CI/CD approach involves a series of stages. Commencing with a product request and a comprehensive plan, the subsequent phases involve coding, building, and subjecting the product to testing [5,68]. When testing is completed, the system will switch from continuous integration to continuous deployment. This starts with the specification of the release, followed by deployment, and then operation—with monitoring the application continuously [69]. This entire sequence is cyclically repeated through the CI/CD pipeline.

The CI/CD pipeline is a series of automated steps for delivering a new version of the software [70]. It is focused on improving software delivery by automating the software development lifecycle. Pipelines are composed of stages, which are composed of steps (commands) to be run in sequence. The stages of a pipeline are built according to some given rules, which depend on the current workflow. Some are run in parallel, while others are run in a sequential order.

Figure 10 shows the general steps of a CI/CD pipeline:

- A clone step that fetches the code from *Git* when a commit is pushed.
- A test step in which smoke, unit, and integration tests are executed.
- A build step that creates a *docker* image and pushes it into a private registry.

- A deploy step that deploys to production if the branch is “master”. If the branch is not “master”, then it deploys to a pre-production environment such as “DEV”, “SIT”, or “UAT”.

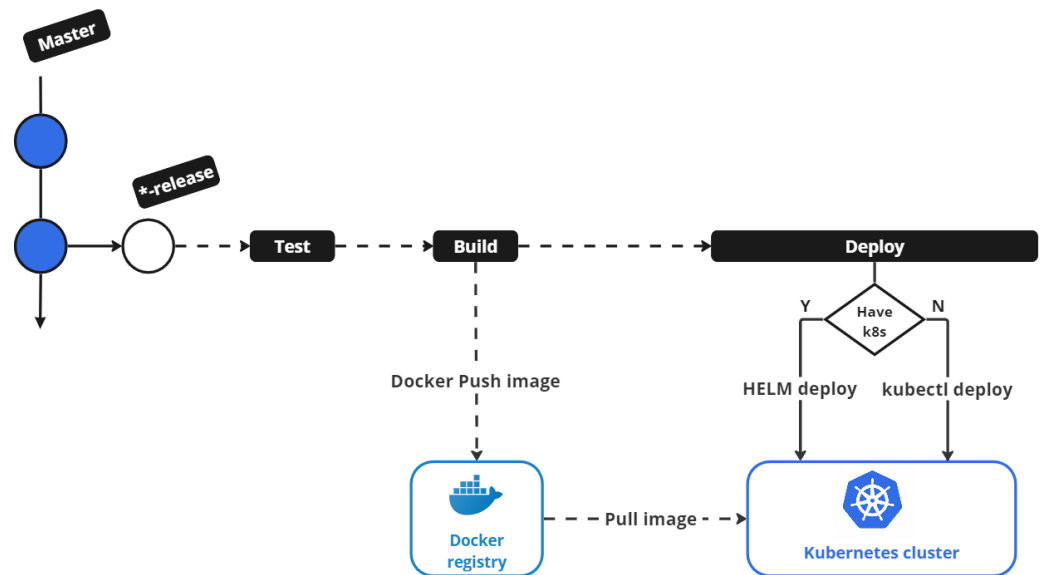


Figure 10. Workflow of CI/CD pipeline in the MPME approach.

The environment lifecycle in the MPME approach should also be managed through the pipeline. A defined set of operations will be run for every action which can be performed on an environment, such as:

- Deploying an environment;
- Destroying an environment;
- Starting an environment;
- Configuring an environment.

To facilitate these actions, we propose an extension to the CI/CD pipeline with the following key functionalities, which have proven instrumental in the successful implementation of the MPME approach:

1. **Integration**—The branch-naming strategy of the MPME approach should be seamlessly integrated into the pipeline operations. This integration ensures the creation of environments without any naming conflicts, thereby enabling unified and easy-to-parameterize access across the pipeline.
2. **Creation of MPME environments**—A dedicated job (e.g., Jenkins job) has been defined exclusively for environment creation. This job will be automatically triggered by the CI/CD pipeline, which allows us to apply the required configuration, bind Kubernetes resources, and provisioning of default secrets within the environment.
3. **Workflow Segregation**—To maintain pipeline clarity and prevent unwieldy complexity, a dedicated workflow should be established for Helm deployment. This approach ensures that the pipeline remains organized and manageable, even as deployment processes evolve.
4. **Modifications**—The CI/CD pipeline should be adapted to accommodate specific scenarios, such as the multi-version deployment of a microservice with different configurations, as well as options for the deployment using Helm or kubectl, and so on. We have provided a script that determines the number of versions requiring deployment for each specific case.
5. **Automatic Deletion**—As part of resource optimization, the MPME approach integrates an automated mechanism for environment cleanup and removal. Upon the successful completion of a project, the related environments and branches are automatically

eradicated. This process involves the comprehensive purging of associated Kubernetes objects, ensuring the systematic removal of the environment from the cluster.

By incorporating these enhancements, the CI/CD pipeline can seamlessly support the robust functionalities and requirements of the MPME approach.

8. Results and Discussion

We have introduced the MPME approach as an augmentation to the existing DevOps infrastructure because it offers a solution that enables seamless concurrent deployment of microservices across various teams and projects, eliminating unnecessary delays between stages. Within the MPME approach, simultaneous code changes and end-to-end tests are fully independent for each project, fostering a more confident and expedited path to production-ready releases.

In the context of the MPME approach, we have introduced the following fundamental concepts for the implementation phases:

- **Automated Environment Provisioning:** To streamline the creation of environments within the existing DevOps infrastructure, we proposed project-based name components for the MPME approach. This concept integrates project-naming components into the branching strategy within the CI/CD pipeline. It helps to enable the automatic creation of multiple development and testing environments simultaneously for each project within a shared cluster.
- **Efficient Resource Allocation:** To optimize resource allocation across multiple projects and microservices, we have proposed the concept of a stable environment called 'PA'. This environment holds the latest post-production releases of the microservices. Rather than each project maintaining its own set of client microservices for testing purposes, a centralized stable 'PA' environment serves as a comprehensive repository. This stable environment enhances resource efficiency by eliminating the necessity to run multiple client microservices within each development environment.
- **Enabling Communication Path:** To establish a communication path between the stable environment (PA) and other development environments, we have adopted the Kubernetes concept of "ExternalName" service. This allows smooth communication between environments, promoting efficient intercommunication and testing.
- **Customized Helm Deployment:** To mitigate the complexities associated with managing and provisioning multiple environments, we introduce a customized Helm chart for the MPME approach. This custom Helm is designed to seamlessly align with our objective of enabling dynamic environment provisioning and configuration. Our refinement of Helm charts involves storing configuration values at two different levels, thus supporting the creation of multiple environments without the need for redundant adjustments to configuration values in multiple places. When a project includes a Helm chart folder, Helm is utilized for packaging, deploying, and managing microservices and Kubernetes resources.
- **Streamlining CI/CD Pipeline:** Throughout the implementation phase of the MPME approach, we ensured the incorporation and refinement of the CI/CD pipeline in alignment with the principles outlined above. This extensive work was focused on optimizing the pipeline to effectively manage the dynamic processes of multiple environments such as their creation, configuration, and decommissioning.

The MPME approach has provided simultaneous development and parallel testing opportunities, as demonstrated in Table 1. This allows multiple deployments of microservices with the same codebase into a single cluster, a capability that was not feasible within the confines of the traditional DevOps infrastructures.

Table 1. Comparison between the MPME approach and the traditional approach.

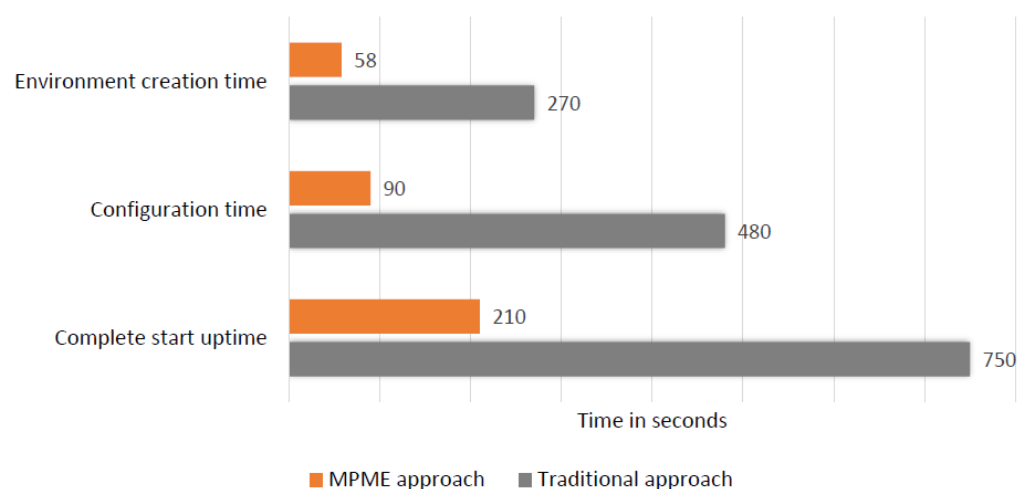
	Traditional Approach	MPME Approach
Environment isolation	Yes	Yes
Simultaneous deployment to single cluster	No	Yes
Availability for parallel testing	No	Yes
Process automation	Yes	Yes
Real-time monitoring	Yes	Yes
Automated environment creation and configuration	No	Yes
Integration with existing DevOps tools	Yes	Yes

8.1. Performance Evaluation

The MPME approach can operate in a fully automatic manner in alignment with the *continuous* nature of DevOps. Therefore, developers do not need to manually manage Dockerfile or Kubernetes manifest files, which enables them to focus more on development tasks. Moreover, the performance of the management of the environment lifecycle is significantly improved with the MPME approach, as discovered in our observation-based measurements. We gathered data from eight DevOps engineers who manually created and configured environments, and then we compared the average time spent on these activities to that of the automatic creation and configuration of environments in the MPME approach.

8.1.1. Measurement on Environment Management

The bar chart in Figure 11 illustrates the comparative time taken for environment creation with essential configurations under two scenarios: the MPME approach and the traditional approach. As seen on the chart, environment creation in the MPME approach takes around 58 s, while using the traditional approach, it takes 4.7 times more (270 s). If the default configurations of the microservices are manually set in the environment, the process will require around 8 min. In contrast, using the MPME approach, we were able to cut that time down to 1.5 min on average. Moreover, the complete startup time of the environment with MPME stands at approximately 4 min, which is a remarkable reduction of 3.5 times in comparison to the traditional approach.

**Figure 11.** Measured time for the environment management.

8.1.2. Resource Consumption

In this subsection, we focus on the resource utilization of both approaches within a computing environment. The primary objective of this analysis is to evaluate the impact of each approach on memory and CPU utilization.

In the traditional approach, resource allocations are determined based on the average number of microservices required to execute a full-cycle test for developed microservices. According to the experimental data of the company participating in our research, each development environment typically required 12 client microservices to support the testing process for code-changed microservices within that environment. Therefore, our calculations are based on the current number of concurrent environments and the average number of the required client microservices, along with their respective resource consumption. In this context, the minimum memory allocation per microservice is currently set at 300 MB, while CPU allocation stands at 0.1 core.

By contrast, the MPME approach introduces the concept of a stable environment designed specifically for full-cycle testing. This stable environment serves as a dedicated repository for post-production deployments, housing the latest stable releases of microservices. As a result, it helps to eliminate the need for excessive resource allocation for multiple overlapping instances of running microservices in various development environments.

The advantages of the MPME approach become apparent when examining resource utilization, as it significantly reduces resource consumption by a factor of 24 in comparison to the traditional approach. To be precise, memory utilization decreases to 3600 MB, a substantial reduction from that of the traditional approach, 86,400 MB. Additionally, CPU utilization is streamlined to 1.2 cores, a significant decrease from the 28.8 cores of the traditional approach. The stable environment within the MPME approach offers efficient resource management in both memory and CPU utilization, making it an ideal choice for clusters with limited capacity, especially in resource-constrained shared environments.

8.2. Scalability of Multi-Environments and Applications

In the MPME approach, the scalability of environments, both in terms of scaling up and down, relies on the availability of cluster resources. Our current implementation of the MPME approach can efficiently support up to 24 concurrent environments, accommodating the company's cluster capacity and ongoing project activities. However, this scalability is not fixed and can be dynamically adjusted to accommodate more concurrent environments if additional cluster capacity becomes available.

To address the scalability of application instances within these environments, we have adopted an autoscaling strategy [71]. We defined specific CPU utilization thresholds for each microservice by leveraging Kubernetes' horizontal autoscaler [72]. When a microservice experiences a significant increase in load, the horizontal autoscaler automatically increases the number of instances by up to three, as required. This ensures that the microservices will be able to smoothly adjust to fluctuating workloads and resource requirements.

8.3. Practical Implementation Challenges

When considering the adoption of the MPME approach, certain challenges may arise. These challenges may require the adaptation of the already existing processes, an efficient orchestration of resource allocation, potential performance overhead associated with containerization, the complexity of managing multiple project requirements, and ensuring compatibility with the already existing tools. To address these challenges effectively and mitigate potential issues, strategies such as automated scaling, performance tuning, comprehensive monitoring, well-defined requirements, and a gradual adoption can be implemented.

Furthermore, the increased number of application containers and the associated Kubernetes resources, especially numerous ExternalName service objects, can have a notable impact on the management pods at the core of the cluster. In our experience, the performance issues encountered were related to service discovery and DNS management due to the proliferation of ExternalName service objects for each environment. Initially, this

led to occasional malfunctions in the CoreDNS [73] pod. Over time, as the project activity increased, this issue escalated, causing the CoreDNS pod to frequently crash and restart due to out-of-memory issues. The pod struggled to handle the high volume of DNS queries with its initial resource allocation.

To address this challenge, we undertook performance-tuning measures. We carefully calculated the required resources and substantially increased the resource allocation for the CoreDNS pod. Additionally, we scaled up the number of CoreDNS pods to five, distributed across the five worker nodes. Moreover, we implemented a strategy to mitigate the load of constant DNS search queries by defining search sequences for DNS queries. A detailed description of this implementation can be found in our recent research paper [74].

It is vital to address potential resource challenges in the cluster resulting from the growing number of applications in multiple environments. We strongly advise defining and assigning the optimal amount of resources for each microservice based on its specific requirements. In this way, we can avoid the allocation of excessive or unnecessary resources, as well as steering clear of relying solely on bare minimum or default resource settings. Over-allocating resources can monopolize cluster capacity with underutilized tasks, whereas under-allocating resources may result in component instability and frequent failures. Achieving the optimal balance and allocating resources according to each microservice's needs are critical for enhancing cluster performance and resource utilization.

9. Conclusions

As the DevOps and CI/CD practices become increasingly prevalent, a multitude of practical improvements and solutions have already emerged to tackle the challenges that arise in various scenarios. This paper proposed the MPME approach, which is designed to handle multiple dynamic environments for the simultaneous deployment of microservices in various and potentially overlapping projects in a single cluster. We have used widespread and popular DevOps tools and containerization technologies to carry out our experiment, but the approach is general enough to accommodate to various circumstances. Besides effectively eradicating bottlenecks within the software development cycle, the MPME approach simplifies and automates the intricate process of creating and managing numerous environments within a self-managed private DevOps infrastructure.

To facilitate the implementation of the MPME approach, we introduced a project-naming component. This component plays a pivotal role in enabling the creation of dynamic environments to support multiple concurrent project activities. It seamlessly integrates into the entire DevOps workflow, encompassing project initiation, branching strategies, version control systems, repositories, and the CI/CD pipeline workflow.

Another noteworthy aspect of our MPME approach is the establishment of a stable environment that consistently hosts the latest releases of the post-production microservices for comprehensive cycle testing. This innovative concept has not only helped us to save cluster resources, but has also paved the way for the development of high-quality production applications. Furthermore, our customized Helm chart streamlines the implementation process, rendering it more efficient, smooth, and well-organized.

Author Contributions: Conceptualization, B.E. and T.K.; methodology, B.E., B.B. and T.B.; supervision, T.K.; visualization, B.E.; writing—original draft, B.E.; writing—review and editing, B.E. and T.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Research data are contained within the article. Extensive technical data are available from the correspondence author with the permission of the case company.

Acknowledgments: We would like to acknowledge and thank the company and its representatives for their contributions and involvement in the study.

Conflicts of Interest: Temuulen Batsuren was employed by Khan Bank. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Bass, L.; Weber, I.; Zhu, L. *DevOps: A Software Architect's Perspective*, 1st ed.; Addison-Wesley Professional: Boston, MA, USA, 2015.
2. Zhu, L.; Bass, L.; Champlin-Scharff, G. DevOps and Its Practices. *IEEE Softw.* **2016**, *33*, 32–34. [CrossRef]
3. Humble, J.; Farley, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 1st ed.; Addison-Wesley Professional: Boston, MA, USA, 2010.
4. Microsoft Azure. DevOps Definition. Available online: <https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-devops/> (accessed on 3 May 2023).
5. Wettinger, J.; Andrikopoulos, V.; Leymann, F. Enabling DevOps Collaboration and Continuous Delivery Using Diverse Application Environments. In *On the Move to Meaningful Internet Systems: OTM 2015 Conferences, Proceedings of the OTM 2015, Rhodes, Greece, 26–30 October 2015*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2015; Volume 9415. [CrossRef]
6. Yu, Y.; Silveira, H.; Sundaram, M. A microservice based reference architecture model in the context of enterprise architecture. In *Proceedings of the 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, Xi'an, China, 3–5 October 2016; pp. 1856–1860. [CrossRef]
7. Villamizar, M.; Garcés, O.; Castro, H.; Verano, M.; Salamanca, L.; Casallas, R.; Gil, S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Proceedings of the 2015 10th Computing Colombian Conference (10CCC)*, Bogota, Colombia, 21–25 September 2015; pp. 583–590. [CrossRef]
8. Kalske, M.; Mäkitalo, N.; Mikkonen, T. Challenges When Moving from Monolith to Microservice Architecture. In *Current Trends in Web Engineering. ICWE 2017*; Garrigós, I., Wimmer, M., Eds.; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2018; Volume 10544. [CrossRef]
9. Bernstein, D. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Comput.* **2014**, *1*, 81–84. [CrossRef]
10. Wan, X.; Guan, X.; Wang, T.; Bai, G.; Choi, B.-Y. Application deployment using Microservice and Docker containers: Framework and optimization. *J. Netw. Comput. Appl.* **2018**, *119*, 97–109. [CrossRef]
11. RedHat. Understanding DevOps Automation. Available online: <https://www.redhat.com/en/topics/automation/what-is-devops-automation> (accessed on 3 May 2023).
12. Armenise, V. Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery. In *Proceedings of the 2015 IEEE/ACM 3rd International Workshop on Release Engineering*, Florence, Italy, 19 May 2015; pp. 24–27. [CrossRef]
13. Bass, L.; Clements, P.; Kazman, R. *Software Architecture in Practice*, 3rd ed.; Addison-Wesley: Westford, MA, USA, 2012.
14. Saito, H.; Lee, H.-C.; Wu, C.-Y. *DevOps with Kubernetes*; Packt Publishing: Birmingham, UK, 2017; ISBN 978-1-78839-664-6.
15. Jenkins, D.; Arnaud, J.; Thompson, S.; Yau, M.; Wright, J. Version Control and Patch Management of Protection and Automation Systems. In *Proceedings of the 12th IET International Conference on Developments in Power System Protection*, Copenhagen, Denmark, 31 March–3 April 2014.
16. Spinellis, D. Git. *IEEE Softw.* **2012**, *29*, 100–101. [CrossRef]
17. Chandrasekara, C.; Herath, P. Branching with Azure Git Repos. In *Hands-On Azure Repos*; Apress: Berkeley, CA, USA, 2020. [CrossRef]
18. Chen, L. Microservices: Architecting for Continuous Delivery and DevOps. In *Proceedings of the 2018 IEEE International Conference on Software Architecture (ICSA)*, Seattle, WA, USA, 30 April–4 May 2018; pp. 39–397. [CrossRef]
19. Shahin, M.; Babar, M.A.; Zhu, L. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* **2017**, *5*, 3909–3943. [CrossRef]
20. Anderson, C. Docker [Software engineering]. *IEEE Softw.* **2015**, *32*, 102–c3. [CrossRef]
21. Rad, B.B.; Bhatti, H.J.; Ahmadi, M. An Introduction to Docker and Analysis of its Performance. *IJCSNS Int. J. Comput. Sci. Netw. Secur.* **2017**, *17*, 228–235.
22. Sayfan, G. *Mastering Kubernetes*, 2nd ed.; Packt Publishing: Birmingham, UK, 2018; ISBN 978-1788999786.
23. The Kubernetes Authors. Kubernetes Official Website. Available online: <https://kubernetes.io/> (accessed on 5 May 2023).
24. Burns, B.; Vohra, A. *Kubernetes: Up and Running: Dive into the Future of Infrastructure*; O'Reilly Media, Incorporated: Sebastopol, CA, USA, 2016.
25. Buchanan, S.; Rangama, J.; Bellavance, N. Helm Charts for Azure Kubernetes Service. In *Introducing Azure Kubernetes Service*; Apress: Berkeley, CA, USA, 2020. [CrossRef]
26. Helm. Charting Our Future—Helm3 Preview. Available online: <https://Helm.sh/blog/Helm-3-preview-pt1/> (accessed on 3 May 2023).
27. Duffy, M. *DevOps Automation Cookbook*; Packt Publishing: Birmingham, UK, 2015.
28. Been, H.; Keiholz, E.; Staal, E. *Azure Infrastructure as Code: With ARM Templates and Bicep*; Manning: Henley-on-Thames, UK, 2022; pp. 239–251.
29. Brikman, Y. *Terraform: Up and Running*; O'Reilly Media: Sebastopol, CA, USA, 2022.
30. Jourdan, S.; Pomes, P. *Infrastructure as Code (IAC) Cookbook*; Packt Publishing: Birmingham, UK, 2017; pp. 217–230.

31. Omofoyewa, Y.; Grebe, A.; Leusmann, P. IaC reusability for Hybrid Cloud Environment. Available online: https://www.researchgate.net/publication/357281177_IaC_reusability_for_Hybrid_Cloud_Environment (accessed on 20 October 2023).
32. Sayfan, G.; Ibryam, B. *Mastering Kubernetes: Dive into Kubernetes and Learn How to Create and Operate World-Class Cloud-Native Systems*; Packt Publishing: Birmingham, UK, 2023; pp. 297–311.
33. Helm Chart Template Guideline. Available online: https://helm.sh/docs/chart_template_guide/ (accessed on 5 May 2023).
34. Butcher, M.; Farina, M.; Dolitsky, J. *Learning Helm: Managing Apps on Kubernetes*; O'Reilly Media, Incorporated: Sebastopol, CA, USA, 2021.
35. Authors of Tarka Labs: Handling Multiple Environments with Helm. Available online: <https://blog.tarkalabs.com/handling-multiple-environments-with-helm-kubernetes-f214192f8f7b> (accessed on 20 October 2023).
36. Performing a Rolling Update. Available online: <https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/> (accessed on 20 October 2023).
37. Sayfan, G. *Hands-On Microservices with Kubernetes: Build, Deploy, and Manage Scalable Microservices on Kubernetes*; Packt Publishing: Birmingham, UK, 2019; pp. 317–338.
38. Qovery. Basic Concepts and Preview Environments. Available online: <https://www.qovery.com/docs/getting-started> (accessed on 3 May 2023).
39. Octopus. Overview. Available online: <https://octopus.com/features> (accessed on 3 May 2023).
40. Machiraju, S.; Gaurav, S. Deployment via TeamCity and Octopus Deploy. In *DevOps for Azure Applications*; Apress: Berkeley, CA, USA, 2018. [CrossRef]
41. Humanitec. Introduction. Available online: <https://developer.humanitec.com/getting-started/pe/introduction/> (accessed on 3 May 2023).
42. BunnyShell. Documents. Available online: <https://documentation.bunnysHELL.com/docs> (accessed on 3 May 2023).
43. GitLab CI. Documents. Available online: <https://docs.gitlab.com/ee/ci/environments/#create-a-dynamic-environment> (accessed on 20 October 2023).
44. Sureshchandra, K.; Shrinivasavadhani, J. Moving from Waterfall to Agile. In Proceedings of the Agile 2008 Conference, Toronto, ON, Canada, 4–8 August 2008; pp. 97–101. [CrossRef]
45. Kim, G.; Humble, J.; Debois, P.; Willis, J.; Forsgren, N. *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*; IT Revolution Press: Portland, OR, USA, 2021. Available online: <https://books.google.hu/books?id=8kRDEAAAQBAJ> (accessed on 15 March 2012).
46. Chacon, S.; Straub, B. Git Branching. In *Pro Git*; Apress: Berkeley, CA, USA, 2014. [CrossRef]
47. Balalaie, A.; Heydarnoori, A.; Jamshidi, P. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Softw.* **2016**, *33*, 42–52. [CrossRef]
48. The Kubernetes Authors. Secrets. Available online: <https://kubernetes.io/docs/concepts/configuration/secret/> (accessed on 5 May 2023).
49. Jayakantha, I. Microservices—Centralized Configuration with Spring Cloud. Available online: <https://medium.com/@ijayakantha/f2a1f7b78cc2> (accessed on 5 May 2023).
50. Sharma, R.; Singh, A. *Getting Started with Istio Service Mesh: Manage Microservices in Kubernetes.*, 1st ed.; Apress: Berkeley, CA, USA, 2020. [CrossRef]
51. The Kubernetes Authors. Service Accounts. Available online: <https://kubernetes.io/docs/concepts/security/service-accounts/> (accessed on 5 May 2023).
52. The Kubernetes Authors. RBAC Good Practices. Available online: <https://kubernetes.io/docs/concepts/security/rbac-good-practices/> (accessed on 8 May 2023).
53. The Kubernetes Authors. Using RBAC Authorization. Available online: <https://kubernetes.io/docs/reference/access-authn-authz/rbac/> (accessed on 5 May 2023).
54. The Kubernetes Authors. Services. Available online: <https://kubernetes.io/docs/concepts/services-networking/service/> (accessed on 5 May 2023).
55. Brewer, E. Kubernetes and the Path to Cloud Native. In Proceedings of the SoCC '15: Proceedings of the Sixth ACM Symposium on Cloud Computing, Kohala Coast, HI, USA, 27–29 August 2015; p. 167. [CrossRef]
56. Google Kubernetes Engine (GKE). Types of Kubernetes Services. Available online: <https://cloud.google.com/kubernetes-engine/docs/concepts/service#types-of-services> (accessed on 5 May 2023).
57. The Kubernetes Authors. Services-ExternalName. Available online: <https://kubernetes.io/docs/concepts/services-networking/service/#externalname> (accessed on 5 May 2023).
58. The Kubernetes Authors. Services-ClusterIP. Available online: <https://kubernetes.io/docs/concepts/services-networking/service/#type-clusterip> (accessed on 5 May 2023).
59. Wang, K.; Wu, S.; Suo, K.; Liu, Y.; Huang, H.; Huang, Z.; Jin, H. Characterizing and optimizing Kernel resource isolation for containers. *Future Gener. Comput. Syst.* **2023**, *141*, 218–229. [CrossRef]
60. Mallett, A. Writing YAML and Basic Playbooks. In *Red Hat Certified Engineer (RHCE) Study Guide*; Apress: Berkeley, CA, USA, 2021; pp. 63–77. [CrossRef]
61. The Helm Authors. Using Helm. Available online: https://helm.sh/docs/intro/using_helm/ (accessed on 5 May 2023).

62. Mike Treadway—FAUN Publication. Helm Charts for More Complex Projects and How to Secure Them. Available online: <https://faun.pub/a1dfde804226> (accessed on 5 May 2023).
63. The Helm Authors. Values Files. Available online: https://helm.sh/docs/chart_template_guide/values_files/ (accessed on 5 May 2023).
64. Nocentino, A.E.; Weissman, B. Storing Persistent Data in Kubernetes. In *SQL Server on Kubernetes*; Apress: Berkeley, CA, USA, 2021. [CrossRef]
65. The Kubernetes Authors. Persistent Volumes. Available online: <https://kubernetes.io/docs/concepts/storage/> (accessed on 5 May 2023).
66. Martin, P. *Kubernetes: Preparing for the CKA and CKAD Certifications*, 1st ed.; Apress: Berkeley, CA, USA, 2021. [CrossRef]
67. Shahin, M.; Zahedi, M.; Babar, M.A.; Zhu, L. An Empirical Study of Architecting for Continuous Delivery and Deployment. *Empir. Softw. Eng.* **2019**, *24*, 1061–1108. [CrossRef]
68. Uphill, T. *DevOps: Puppet, Docker and Kubernetes*; Packt Publishing: Birmingham, UK, 2017; ISBN 978-1788297615.
69. Arundel, J.; Domingus, J. *Cloud Native DevOps with Kubernetes: Building, Deploying, and Scaling Modern Applications in the Cloud*; O'Reilly Media: Newton, MA, USA, 2019; ISBN 978-1492040767.
70. Arachchi, S.; Perera, I. Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management. In Proceedings of the 2018 Moratuwa Engineering Research Conference (MERCon), Moratuwa, Sri Lanka, 30 May–1 June 2018; pp. 156–161. [CrossRef]
71. Rajput, D.; Rajesh, R.V. *Building Microservices with Spring: Master Design Patterns of the Spring Framework to Build Smart, Efficient Microservices*; Packt Publishing: Birmingham, UK, 2018; pp. 434–437.
72. Nguyen, T.-T.; Yeom, Y.-J.; Kim, T.; Park, D.-H.; Kim, S. Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration. *Sensors* **2020**, *20*, 4621. [CrossRef] [PubMed]
73. CoreDNS: DNS and Service Discovery. Available online: <https://coredns.io/> (accessed on 20 October 2023).
74. Erdenebat, B.; Bud, B.; Kozsik, T. Challenges in service discovery for microservices deployed in a Kubernetes cluster—A case study. *Infocommun. J.* **2023**, *15*, 69–75. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.